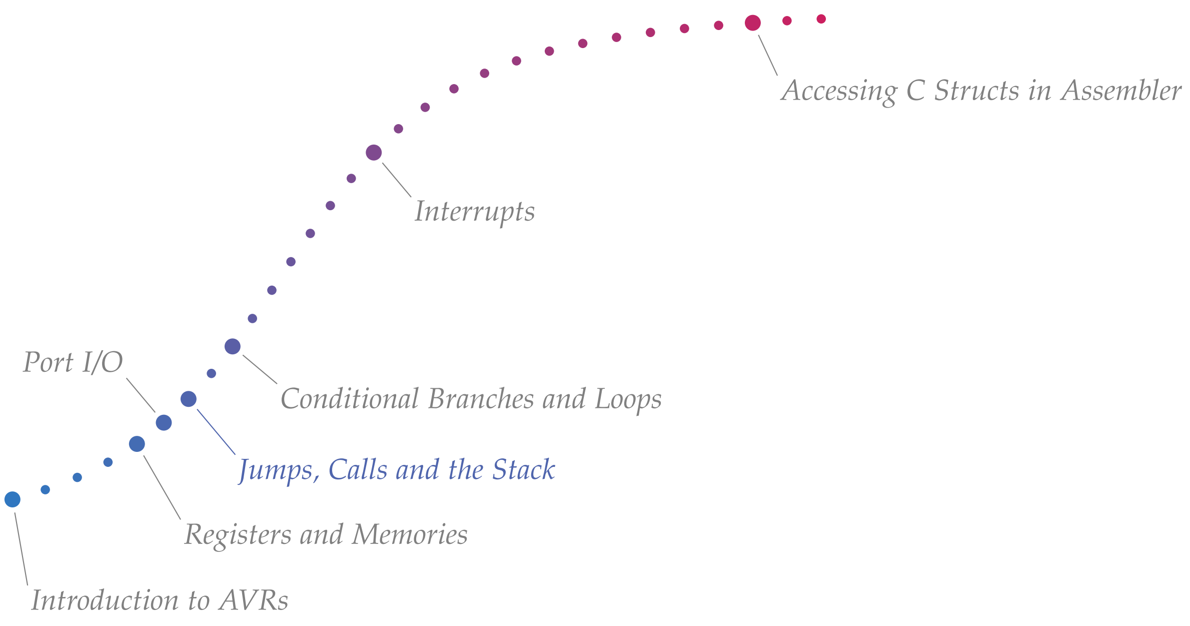
www.avrbeginners.net

Assembler Tutorial

Jumps, Calls and the Stack

Author: Christoph Redecker
Version: 1.0.2


Accessing C Structs in Assembler
Interrupts
Port I/O
Conditional Branches and Loops
Jumps, Calls and the Stack
Registers and Memories
Introduction to AVRs

Basically every assembler program needs one of the different *jump* types. There are a number of different ones, and each will be described in this tutorial. In other (higher–level) programming languages, the `goto` statement is a kind of jump. Jumps are a one–way facility: the program can not return from the code that it jumped to, unless the return location is known.

Downloading the AVR Instruction Set Manual from Atmel's homepage is highly recommended. Its content is also included in the AVR Studio Help.

Another facility usually found in programming languages is the *subroutine* (or function, or procedure, or . . .). Subroutines can be *called*, and when they *return*, the program flow continues with the instruction following the call. The return address is, in contrast to a jump, known. When the subroutine is called, the return address (the address of the instruction *following* the call) is pushed onto the *stack*. When the subroutine returns, the return address is popped from the stack and, effectively, jumped to.

The stack can not only be used for saving and restoring return addresses, but also for other data. A compiler can, for example, use the stack for passing function arguments. This is usually done when their size exceeds the number of registers available for this purpose (a structure whose size is 100 bytes cannot be stored in 32 registers).

Jump Instructions

There are four different jump instructions, and each has of them has special features. The most unsurprising one is `jmp` (*jump*). It is accompanied by the shorter `rjmp` (*relative jump*), `ijmp` (*indirect jump*) and `eijmp` (*extended indirect jump*). The indirect jumps are a bit more complicated, but very powerful. So let's start with `jmp`.

Not all jump instructions are available in all devices — you need to check in the device's datasheet!

Jump

The `jump` instruction jumps to an *absolute* address in program memory. The destination is stored as a 22–bit value, which means that it can jump to any location in a device that has as much as 4M words of program memory. Here is an example of its usage:

```
.org 0x00
jmp main
...  // lots of space for subroutines and other code
.org 0x900  // set program counter to address 0x900
main:       // this is the application's main routine
  rjmp main // and it's just an endless loop.
```

In this example, a `jmp` instruction is placed at the reset vector ($0x00$), and it is used to jump to the application's main routine, which is labeled `main`. It has been placed at address $0x900$ using the `.org` directive.

`.org` itself is not a jump instruction, but and *assembler directive*. It tells the assembler to place the following expressions at a specific address.

The jump destination is stored as an *absolute* address: when the above example is assembled, the value $0x900$ is stored together with the opcode in 2 words (4 bytes) of program memory. This is one of the disadvantages of `jmp`. The other one is that it needs 3 CPU cycles for execution.

There are two reasons to provide a jump instruction with somewhat limited capabilities compared with `jmp`: normal AVRs don't have that much memory, and most jumps destinations are within a close region around the address where the jump is stored. The *relative jump* takes advantage of this situation.

Relative Jump

The destination address of `rjmp` is not stored as an absolute value, but as the difference between the jump destination and the storage location *following the jump*. This is best explained with a few examples.

In the code snippet accompanying the explanation of `jmp` (see above), `rjmp` is used to construct an endless loop:

```
...
.org 0x900
main:
  rjmp main // same as rjmp -1
```

The instruction following the `rjmp` would be at address $0x901$. The jump destination is $0x900$, so the relative jump length is $0x900 - 0x901 = -1$. Again: *repeating* the same instruction over and over is stored as a *jump backwards*. Consequently, a relative jump of zero words as in

```
rjmp 0
```

or

```
here: rjmp there // same as rjmp 0!
there: ...
```

is similar to a `nop`, but needs two CPU cycles for execution. This is one cycle less that `jmp` needs. Also, the jump length for `rjmp` is limited to $\pm$2K words, which means that it can be stored in just one word of program memory. In devices with not more than 4K words (8K bytes), `rjmp` can address the entire memory region, because a relative jump can wrap around the end of

How to choose between `jmp` and `rjmp`

program memory. *You do not need* `jmp` *in devices with 8K bytes of program memory or less*!

In bigger devices, use `rjmp` first and, if the assembler complains about a "relative jump destination out of reach", replace it with `jmp`. Finding places where you can switch back from `jmp` to `rjmp` is not that easy, and if you don't encounter any speed problems, don't look for these opportunities. As tempting as it might be, it is a waste of time.

The two jump instructions that were introduced are only usable in situations where the destination is known at assembly time. In some situations, this is not enough, because the destination is computed at runtime. There is a solution for this: *indirect jumps*.

Indirect Jump

The `ijmp` (*indirect jump*) instruction uses the value stored in the Z register pair as destination address:

```
...
ldi ZL, low(main)
ldi ZH, high(main)
ijmp
...
.org 0x900
main:
  rjmp main
```

First of all, the Z register pair is a *pair*, consisting of the two registers ZL and ZH. They have 16 bits in total, so the pair can store 16–bit addresses. `ijmp` interprets the address in Z as a *word* address, which means that `ijmp` can jump to the first 64K words (128K bytes) of program memory. This is enough for, say, an ATmega128. In the example above, the address of `main` is split into its low byte and high byte (using the assembler functions `low()` and `high()`). These are loaded into ZL and ZH, respectively, to be interpreted by the following `ijmp`. It will jump to `main`. Note that the address of `main` is known in this example, so we haven't gained anything yet! However, we will get back to that later.

The indirect jump needs two clock cycles for execution, which is just as much as `rjmp` needs and one cycle less than `jmp`. However, ZL and ZH must be set prior to executing `ijmp`, which in this case needs an additional 2 clock cycles. The setup needs even more clock cycles when the jump destination must be calculated — which is, at the same time, the true advantage of indirect addressing: the jump destination doesn't need to be known at assembly time. It is possible to alter Z before executing the `ijmp`:

```
...
clr r0
ldi ZL, low(jumpTable)
ldi ZH, high(jumpTable) // Z now points at jumpTable
add ZL, r16
adc ZH, r0 // add value of r16 to Z
ijmp
...
jumpTable:
  rjmp isZero
  rjmp isOne
  rjmp isTwo
  ...
```

The jump destination is now calculated at runtime, from the base address of the jump table and the contents of R16. If we have R16$= 2$, and R1$= 0$, the above program will execute the code stored at `isTwo`. In this sense, it is equivalent to

The use of conditional branches is discussed in more detail in the "Conditional Branches and Loops" tutorial.

```
...
cpi r16, 0
breq isZero
cpi r16, 1
breq isOne
cpi r16, 2
breq isTwo
...
```

Otherwise, there are significant differences: checking every value individually takes time, and the more comparisons are made, the more time is wasted before a "late" comparison is made. It takes more time for the program to reach `isTwo` than for `isZero`. The `ijmp` variant needs the same number of clock cycles for every alternative and is faster when many alternatives are possible!

Another difference is that conditional branches like `breq` cannot jump to far destinations; they can only do a relative jump to destinations in the range $-63 \ldots + 64$ words. This problem can be solved by a different structure of the comparisons, but that results in slower code:

`PC` is an assembler operand and contains the current value of the program counter.

```
...
cpi r16, 0
brne PC+2   // branches the next cpi
rjmp isZero // this is at PC+1
cpi r16, 1  // and this at PC+2
```

```
brne PC+2
rjmp isOne
cpi r16, 2
brne PC+2
rjmp isTwo
...
```

If `jmp` was used in this example, the branches would skip one more word, as in `brne PC+3`.

Again, it is possible that the capabilities of `ijmp` are not enough. Some AVRs have more than 128K bytes of program memory, so `ijmp` can only reach the lower part of program memory in those devices. In those devices, it can be extended.

Extended Indirect Jump

The *extended indirect jump* works just like `ijmp`, but the Z register pair is extended by the EIND register in the I/O space, which acts as a third byte to ZL and ZH:

`eijmp` is only relevant for devices with more than 128K bytes of program memory, like the ATmega1280.

```
...
clr r0
ldi ZL, low(jumpTable)
ldi ZH, high(jumpTable)
ldi r16, byte3(jumpTable) // now: Z -> jumpTable
add ZL, r17 // r17 = offset in jumpTable
adc ZH, r0
adc r16, r0 // third byte for the calculation
out EIND, r16
eijmp
...
.org 0x10000 // can't get here with ijmp!
jumpTable:
  rjmp isZero
  rjmp isOne
  rjmp isTwo
```

Again, the offset used for addressing a routine in the jump table is stored in R17. R0 is used as a zero register, and R17 as a third byte in the calculation of the destination address.

Call Instructions

Call instructions leave a return address on the stack (the stack will be explained in the next section) and jump to the call destination. The jump part of this works exactly like the jumps explained in the previous section, and there are indeed the variants `call` (*call subroutine*), `rcall` (*relative call subroutine*), `icall` (*indirect call to Z*) and `eicall` (*extended indirect call to Z*)

The stack pointer (SP) needs to be set up prior to any subroutine calls. In newer AVRs this is done automatically when the device is reset; on older AVRs this has to be done in the initialisation code:

```
.org 0
rjmp reset
...
reset:
ldi r16, low(RAMEND)
out SPL
ldi r16, high(RAMEND)
out SPH, r16
...
```

`RAMEND` is defined in the device's include file. These come with AVR Studio.

The stack can be used by call instructions to store the return address, which is the address following the call instruction. In devices with 128K bytes of program memory or less, two bytes are stored on the stack (or *pushed onto the stack*). In devices with more program memory, three bytes are needed for a return address.

Call Subroutine

As already mentioned, calls work just like jumps, but the subroutine that is called can return to the calling code:

```
...
call someSubroutine
cbi PORTD, 0
...
someSubroutine:
  sbi PORTD, 0
ret
```

In this example, `someSubroutine` is called: the return address (where `cbi` is stored) is pushed onto the stack, and the CPU jumps to the destination address. The subroutine sets an I/O line and then returns, using `ret`. The return address is popped from the stack and jumped to. The calling code then clears the I/O pin again.

The destination address for `call` is stored as an absolute 22–bit address. The addressing works like the addressing used for `jmp`.

Relative Call Subroutine

The `rcall` instruction is the equivalent to `rjmp`, but as a subroutine call. The destination address is stored as a relative offset from the current address in program memory. However, *the return address is stored in the same way as for* `call`: it is a 16– or 22–bit value on the stack. This means that `ret` can be used for returning from a subroutine that has been `rcall`'ed. In fact, subroutines have no information about how they were called, so all subroutines must use the same rules for returning:

```
...
rcall someSubroutine // relative this time
cbi PORTD, 0
...
someSubroutine:
  sbi PORTD, 0
ret // same return instruction as above
```

The relative call is one word smaller and one CPU cycle faster then the absolute call. Again, try to use `rcall` in big devices and substitute with `call` where necessary.

(Extended) Indirect Call Subroutine

These instructions are the calling equivalents to `ijmp` and `eijmp`. They can used for indirect addressing (jump tables or other trickery) like the jump intructions, and the usage is absolutely not surprising:

```
...
ldi ZL, low(someRoutine)
ldi ZH, high(someRoutine)
icall
... // someRoutine will return to here
.org 0x900
someRoutine:
  ... // whatever this is supposed to do
ret
```

Arguments and Return Values

Usually, a function takes one or more *arguments*, and has some kind of *return value* as well. When the function is called, it needs to know where to look for these arguments, and the calling code needs to know where the function stores the return value. Libraries can define and reuse a set of *calling conventions*, which determine where arguments and return values are stored.

The calling conventions of the popular AVR-libc can be viewed at `http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_reg_usage`

The same thing can be done in assembler, with no restrictions at all (like only having one return value, as in C). A simple example:

```
...
```

```
ldi r16, 0xAA
rcall setLEDs
...
setLEDs:
  com r16
  out PORTB, r16
ret
```

One convention here is that the function's argument is stored in R16. Additionally, it has no return value, and does *not* preserve the value stored in R16. This is part of the calling conventions, and the calling code must save R16 (on the stack, for example) if the value is needed again after the routine has returned. At the very least, document them in some kind of function header:

Having good calling conventions can make your code faster. Having bad calling conventions can slow it down. Having no calling conventions at all usually breaks your code and your plans for the weekend.

```
...
//*******************************************
// setLEDs
// arguments: LEDs to switch on are set in r16
// returns: nothing
// changes: r16
// (LEDs are between PORTB and VCC)
//*******************************************
setLEDs:
  com r16
  out PORTB, r16
ret
```

Additional arguments and return values can be stored in more registers, or on the stack. If you *want* to use the stack for arguments, be warned: it requires some trickery. If you really *need* to use the stack for function arguments, think about redesigning your API. That's why this usage of the stack is not covered by this tutorial.

The Stack

The stack is used for storing return addresses, registers that need to be preserved while some code is changing them, and — if necessary — function arguments. The stack is part of the AVR's SRAM.

General Functionality

There are two important parts of the stack: The Stack Pointer (SP) and the stack space, which is part of the SRAM. This means that the SRAM is shared between your `.data` sections and the stack. If all the available SRAM is used by `.data` sections, the stack cannot be used. If it is used nonetheless, it would corrupt your data. This is known as a "stack collision".

In AVRs, the stack *grows downwards*. This means that higher memory addresses are used first, and the stack pointer points at a location *below* those locations that make up the stack space. This is illustrated in the figure to the right.

	Address	SRAM contents
		...
	X	data[0]
		...
	$X-(n-1)$	data[$n-1$]
SP $\rightarrow$	$X-n$	unused
		...

Figure 1: Stack space and stack pointer (SP): n data bytes are stored on the stack, starting at SRAM address X, and the stack pointer points at the highest unused address.

The two basic stack operations are `push` and `pop`. When data is pushed onto the stack, it is stored at the address the stack pointer is currently pointing at. The stack pointer is *decremented* afterwards. This is exactly how the stack grows downwards. Not it should also be clear why the stack pointer is usually initialised with *RAMEND*: it then grows downwards from the highest available memory location.

When data is popped from the stack, the stack pointer is first *incremented*, and then the data is stored in the `pop` operation's destination register. Here is an example showing both operations:

```
ldi r16, low(0x200)
out SPL, r16
ldi r16, high(0x200)
out SPH, r16
push r16
// r16 is stored at 0x200, SP is now 0x1FF
ldi r16, 0xFF // or more code ...
pop r16 // overwrites r16, which was 0xFF
// SP is now 0x200 again
...
```

This is how registers can be preserved while other routines are changing them. *Interrupt service routines* (ISRs) make use of this technique to preserve the contents of SREG:

You can also reserve one register for preserving the SREG. This will speed up most ISRs, but you have one register less in your main code.

```
push r16
in r16, SREG
push r16
... // actual ISR code
pop r16
out SREG, r16
pop r16
reti
```

`reti` is not a typo. It returns from a subroutine, and sets the global interrupt enable bit in SREG. It is usually only used in ISRs.

Return Addresses

To the stack, return addresses are just like any other kind of data. The return address is pushed onto the stack when a subroutine is called, and the stack pointer is set to point at the highest free location in SRAM. A normal subroutine call

```
call mySubroutine
...
mySubroutine:
  ... // subroutine code
  ret
```

can be taken apart into the following steps:

```
ldi r16, low(PC+6) // address of the nop (below)
push r16
ldi r16, high(PC+4)
push r16
jmp mySubroutine
nop
...
mySubroutine:
  ... // subroutine code
  ret
```

The low byte of the return address is actually pushed onto the stack first. This can be seen in the simulator.

In this piece of code, the loy byte of the address of the `nop` (which is 6 words ahead of the first `ldi`) is pushed on the stack, followed by the high byte (the `nop` is 4 words ahead of the second `ldi`). The the code jumps to the subroutine, and the subroutine uses `ret` to return — using the return address that was pushed on the stack before, just like if the code used `call` or `rcall` before.

This actually works, but uses a lot more CPU cycles and program memory than the call instructions. However, the `ret` instruction can also be dissected in a similar way:

```
mySubroutine:
  ... // subroutine code
  pop ZH
  pop ZL
  ijmp
```

Again, note that the subroutine does not know where it was called from, and must consequently use an indirect jump here. Also, the Z register pair is altered, which is probably not desired, and doesn't happen when the `ret` instruction is used.